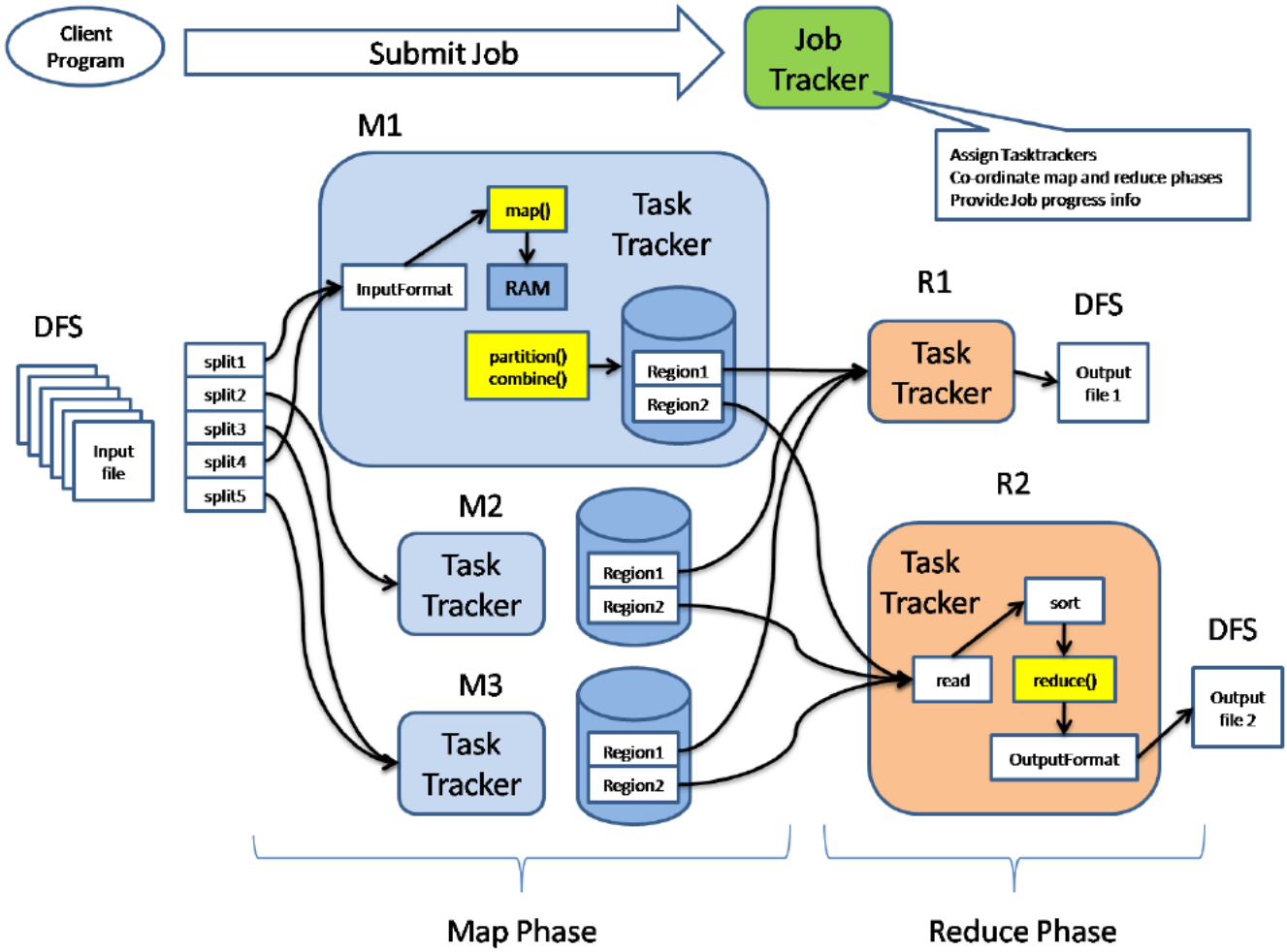
Jamzy Wang

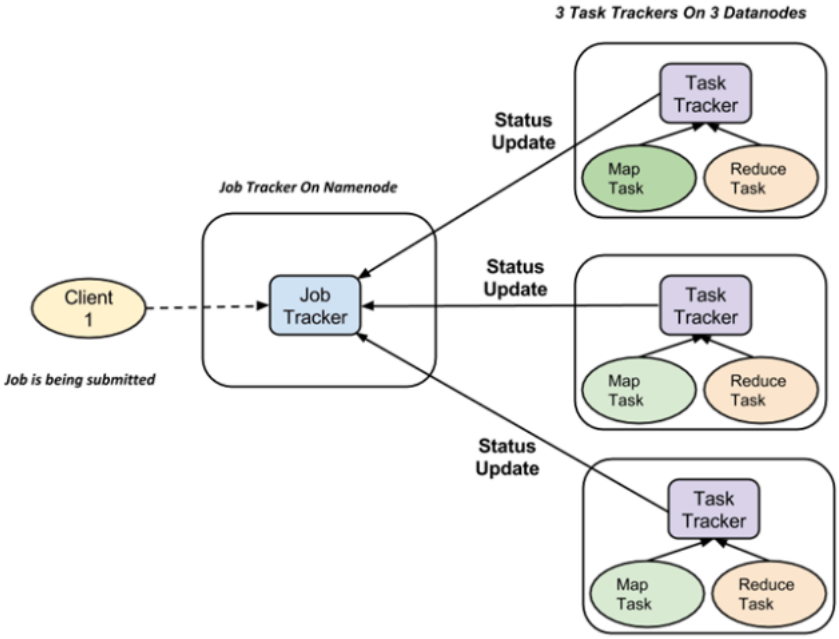
Hadoop Map/Reduce执行流程详解

2015-05-19 21:40

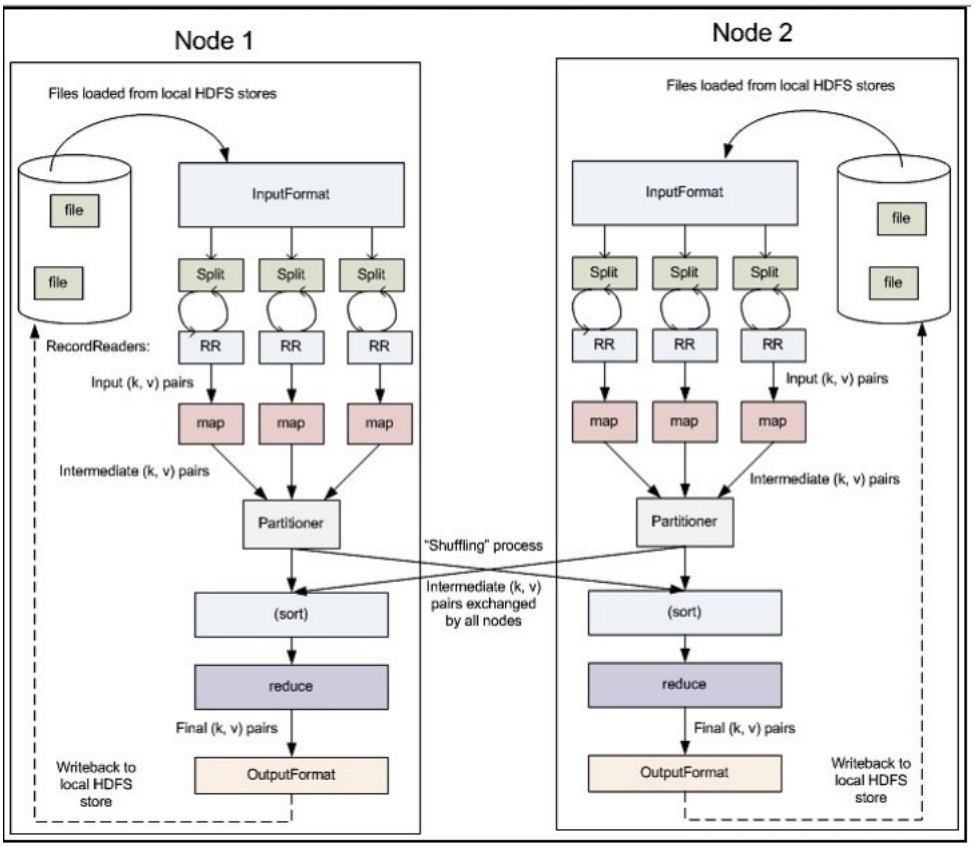
一个Map/Reduce 作业（job）通常会把输入的数据（input file）切分为若干独立的数据块（splits），然后由 map任务（task）以完全并行的方式处理它们。Map/Reduce框架会对map的输出做一个 Shuffle 操作，Shuffle 操作的后的结果会输入给reduce任务。整个Map/Reduce框架负责任务的调度和监控，以及重新执行已经失败的任务。



Map/Reduce计算集群由一个单独的JobTracker（master）和每个集群节点一个 TaskTracker（slave）共同组成。JobTracker负责调度构成一个作业的所有任务，这些任务会被分派到不同的TaskTracker上去执行，JobTracker会监控它们的执行、重新执行已经失败的任务。而TaskTracker仅负责执行由JobTracker指派的任务。

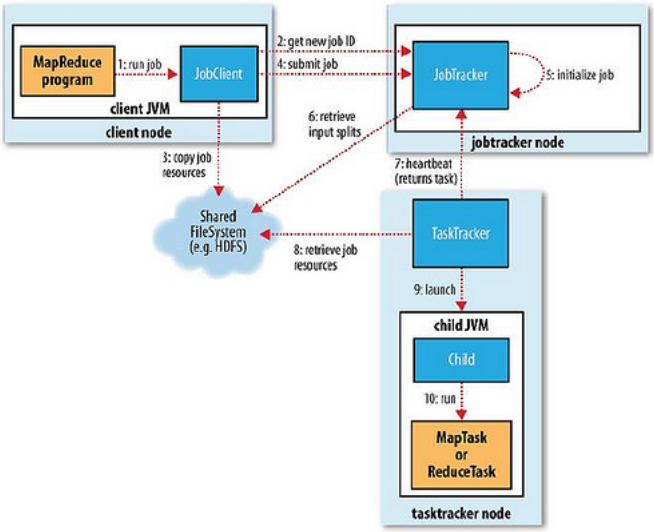


本文将按照map/reduce执行流程中各个任务的时间顺序详细叙述map/reduce的各个任务模块，包括：输入分片（input split）、map阶段、combiner阶段、shuffle阶段和reduce阶段。下图是一个不错的执行流程图：



作业的提交与监控

JobClient是用户提交的作业与JobTracker交互的主要接口。



JobClient提交作业的过程如下：

(1) map/reduce程序通过runJob()方法新建一个JobClient实例;

(2) 向JobTracker请求一个新jobID，通过JobTracker的getNewJobId()获取；

(3) 检查作业输入输出说明。如果没有指定输出目录或者输出目录已经存在，作业将不会被提交，map/reduce程序； 输入作业划分split，如果划分无法计算（如：输入路径不存在），作业将不会被提交，错误返回给map/reduce程序。

(4) 将运行作业所需要的资源（作业的jar文件、配置文件、计算所得的输入划分）复制到一个以作业ID命名的目录中；

(5) 通过调用JobTracker的submitJob()方法，告诉JobTracker作业准备提交；

(6) JobTracker将提交的作业放到一个内部队列中，交由作业调度器进行调度，并对其进行初始化。

(7) 创建Map任务、Reduce任务：一个split对应一个map，有多少split就有多少map; Reduce任务的数量由JobConf的mapred.reduce.tasks属性决定

(8) TaskTracker执行一个简单的循环，定期发送心跳（heartbeat）给JobTracker

Input files

Input file是map/reduce任务的原始数据，一般存储在HDFS上。应用程序至少应该指明输入/输出的位置（路径），并通过实现合适的接口或抽象类提供map和reduce函数。再加上其他作业的参数，就构成了作业配置（job configuration）。然后，Hadoop的 job client提交作业（jar包/可执行程序等）和配置信息给JobTracker，后者负责分发这些软件和配置信息给slave、调度任务并监控它们的执行，同时提供状态和诊断信息给job-client。

InputFormat

InputFormat为Map/Reduce作业输入的细节规范。Map/Reduce框架根据作业的InputFormat来：

(1) 检查作业输入的正确性，如格式等。

(2) 把输入文件切分成多个逻辑InputSplit实例， 一个InputSplit将会被分配给一个独立的Map任务。

(3) 提供RecordReader实现，这个RecordReader从逻辑InputSplit中获得输入记录（“ K-V对” ），这些记录将由Map任务处理。

InputFormat有如下几种:

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

InputFormats provided by MapReduce

- **TextInputFormat**:

TextInputFormat是默认的INputFormat，输入文件中的每一行就是一个记录，Key是这一行的byte offset，而value是这一行的内容。如果一个作业的Inputformat是TextInputFormat，并且框架检测到输入文件的后缀是 `.gz` 和 `.lzo`，就会使用对应的CompressionCodec自动解压缩这些文件。但是需要注意，上述带后缀的压缩文件不会被切分，并且整个压缩文件会分给一个mapper来处理。

- **KeyValueTextInputFormat**

输入文件中每一行就是一个记录，第一个分隔符字符切分每行。在分隔符字符之前的内容为Key，在之后的为Value。分隔符变量通过key.value.separator.in.input.line变量设置，默认为(\t)字符。

- **NLineInputFormat**

与TextInputFormat一样，但每个数据块必须保证有且只有N行，mapred.line.input.format.linespermap属性，默认为1。

- **SequenceFileInputFormat**

一个用来读取字符流数据的InputFormat，为用户自定义的。字符流数据是Hadoop自定义的压缩的二进制数据格式。它用来优化从一个MapReduce任务的输出到另一个MapReduce任务的输入之间的数据传输过程。

InputSplits

InputSplit是一个单独的Map任务需要处理的数据块。一般的InputSplit是字节样式输入，然后由RecordReader处理并转化成记录样式。通常一个split就是一个block，这样做的好处是使得Map任务可以在存储有当前数据的节点上运行本地的任务，而不需要通过网络进行跨节点的任务调度。

可以通过设置mapred.min.split.size， mapred.max.split.size, block.size来控制拆分的大小。如果mapred.min.split.size大于block size，则会将两个block合成到一个split，这样有部分block数据需要通过网络读取；如果mapred.max.split.size小于block size，则会将一个block拆成多个split，增加了Map任务数。

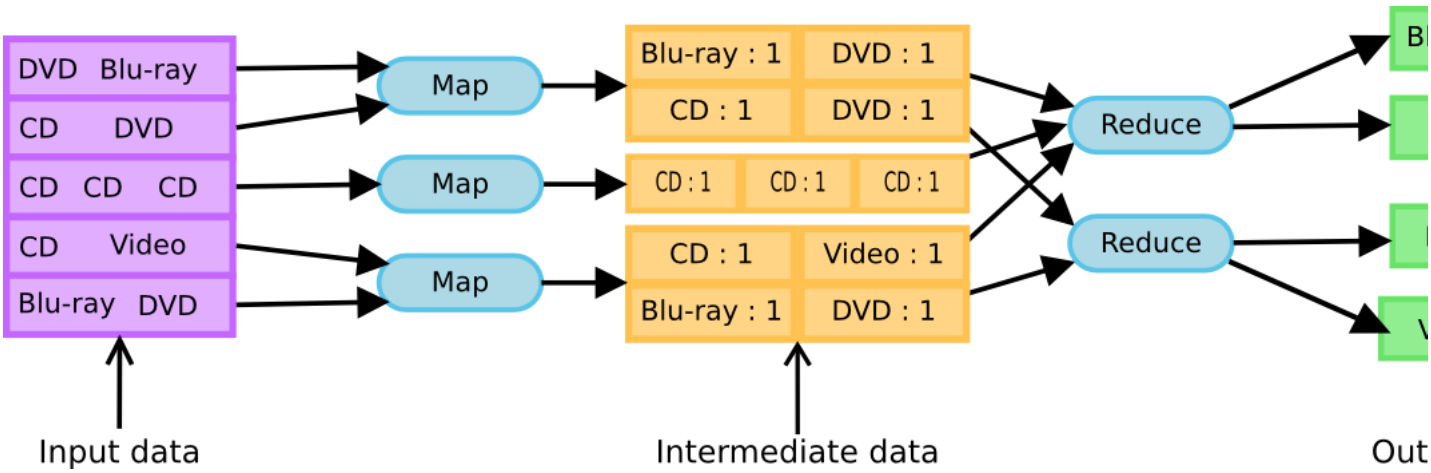
假设splitSize是默认的64M，现在输入包含3个文件，这3个文件的大小分别为10M，64M，100M，那么这3个文件会被分割为：

```
1 输入文件大小                  10M     64M     100M
2 分割后的InputSplit大小        10M     64M     64M，36M
```

在Map任务开始前，会先获取文件在HDFS上的路径和block信息，然后根据splitSize对文件进行切分（splitSize = computeSplitSize(blockSize, minSize, maxSize) ），默认splitSize 就等于blockSize的默认值（64m）。

Mapper

Map是一类将输入记录集转换为中间格式记录集的独立任务，主要是读取InputSplit的每一个Key,Value对并进行处理



确定map任务数量

Map/Reduce框架为每一个InputSplit产生一个map任务，而每个InputSplit是由该作业的InputFormat产生的,默认一个InputSplit大小就等于blockSize的默认值。因此，maps的数量通常取决于输入大小,也即输入文件的block数。 因此，假如输入数据有10TB，而block大小为64M，则需要164,000个map。map正常的并行规模大致是每个节点（node）大约10到100个map，对于CPU 消耗较小的map任务可以设到300个左右。

因为启动任务也需要时间，所以在一个较大的作业中，最好每个map任务的执行时间不要少于1分钟，这样可以让启动任务的开销占比尽可能的低。对于那种有大量小文件输入的的作业来说，一个map处理多个文件会更有效率。如果输入的是打文件，那么一种提高效率的方式是增加block的大小（比如512M），每个map还是处理一个完整的HDFS的block。

当在map处理的block比较大的时候，确保有足够的内存作为排序缓冲区是非常重要的，这可以加速map端的排序过程。假如大多数的map输出都能在排序缓冲区中处理的话应用的性能会有极大的提升。这需要运行map过程的JVM具有更大的堆。

网格模式：确保map的大小，使得所有的map输出可以在排序缓冲区中通过一次排序来完成操作。

合适的map数量有以下好处：

(1) 减少了调度的负担；更少的map意味着任务调度更简单，集群中可用的空闲槽更多。

(2) 有足够的内存将map输出容纳在排序缓存中，这使map端更有效率；

(3) 减少了需要shuffle map输出的寻址次数，每个map产生的输出可用于每一个reduce，因此寻址数就是map个数乘以reduce个数；

(4) 每个shuffle 的片段更大，这减少了建立连接的相对开销，所谓相对开销是指相对于在网络中传输数据的过程。

(5) 这使reduce端合并map输出的过程更高效，因为合并的次数更少，因为需要合并的文件段更少了。

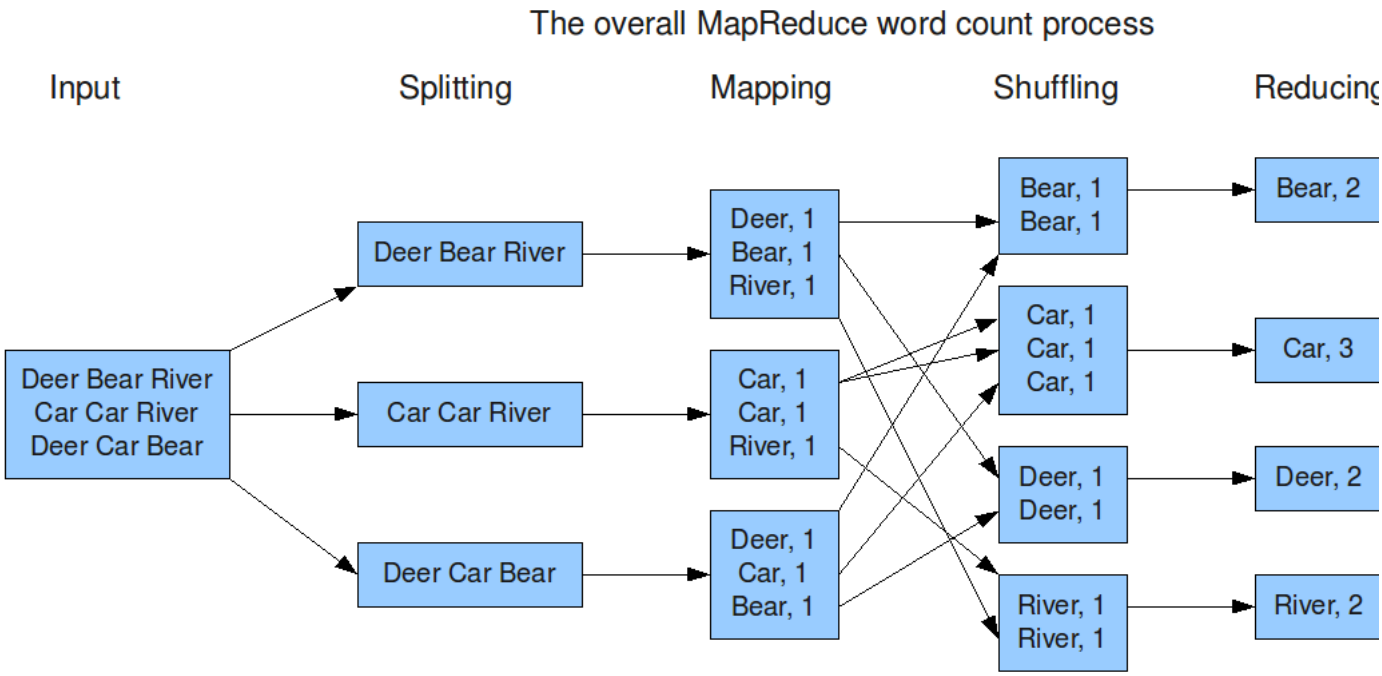
执行map任务

Mapper的实现者需要重写 JobConfigurable.configure(JobConf)方法，这个方法需要传递一个JobConf参数，目的是完成Mapper的初始化工作。然后，**框架为这个任务的InputSplit中每个键值对调用一次 map**(WritableComparable, Writable, OutputCollector, Reporter)操作。这里需要指出很

多人的一种错误认识——“输入和输出的键值对类型一致，一一对应”，这种认识是错误的。输入输出键值对的关系如下：

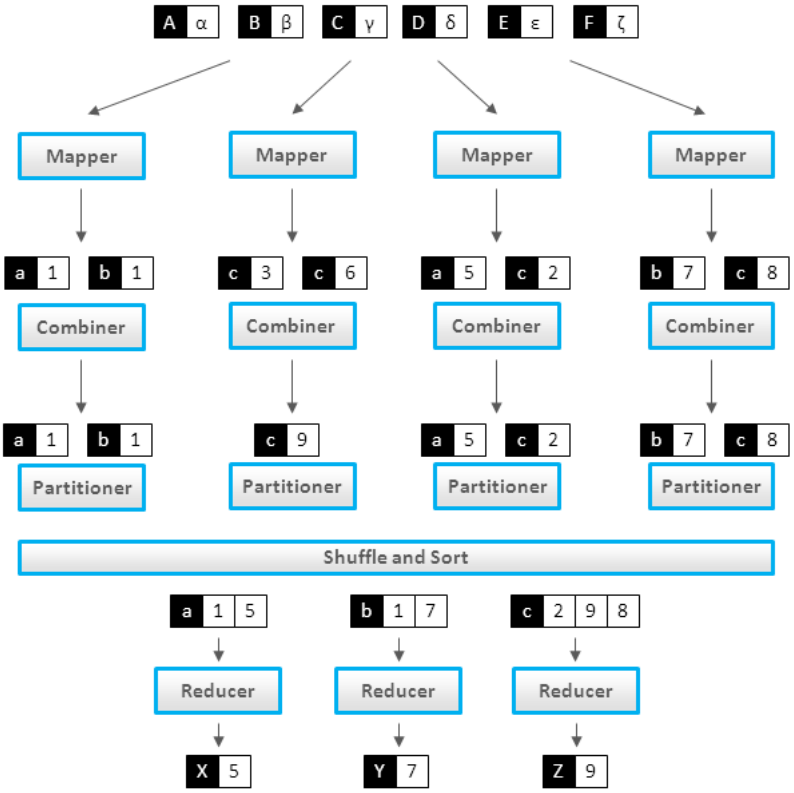
1 1）输出键值对不需要与输入键值对的类型一致。
2 2）一个给定的输入键值对可以映射成0个或多个输出键值对。

以 word count为例，输入不需要是“一行个单词”的形式，可以是一行许多个单词，输入一行可以对应多行输出，如下图所示：



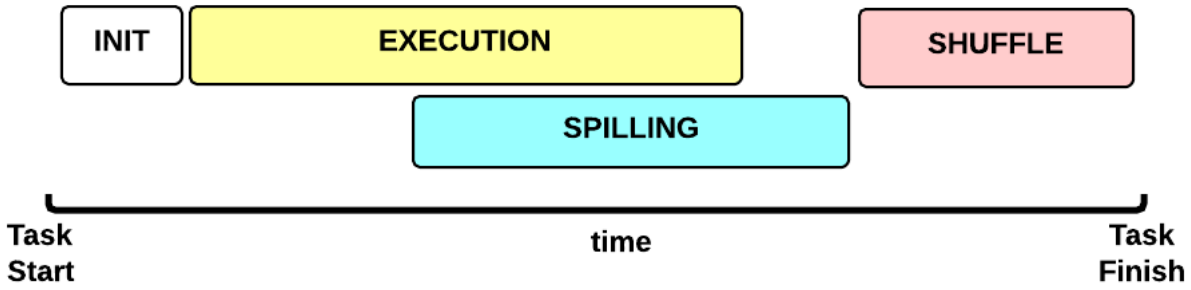
通过调用OutputCollector.collect(WritableComparable,Writable)可以收集map(WritableComparable, Writable, OutputCollector, Reporter)输出的键值对。应用程序可以使用Reporter报告进度，设定应用级别的状态消息，更新Counters（计数器），或者仅是表明自己运行正常。

Map/Reduce框架随后会把与一个特定key关联的所有中间过程的值（value） 分组 并 排序 这个分组和排序过程被称为Shuffle,然后把它们传给Reducer以产出最终的结果。**分组的总数目和一个作业的reduce任务的数目是一样的。**用户可以通过实现自定义的 Partitioner来控制哪个key被分配给哪个 Reducer。 对于map的输出，用户可选择通过JobConf.setCombinerClass(Class)指定一个combiner，它负责对中间过程的输出进行本地的聚集，这会有助于降低从Mapper到 Reducer数据传输量。



这些被排好序的中间过程的输出结果保存的格式是(key-len, key, value-len, value)，应用程序可以通过JobConf控制对这些中间结果是否进行压缩以及怎么压缩，使用哪种CompressionCodec。

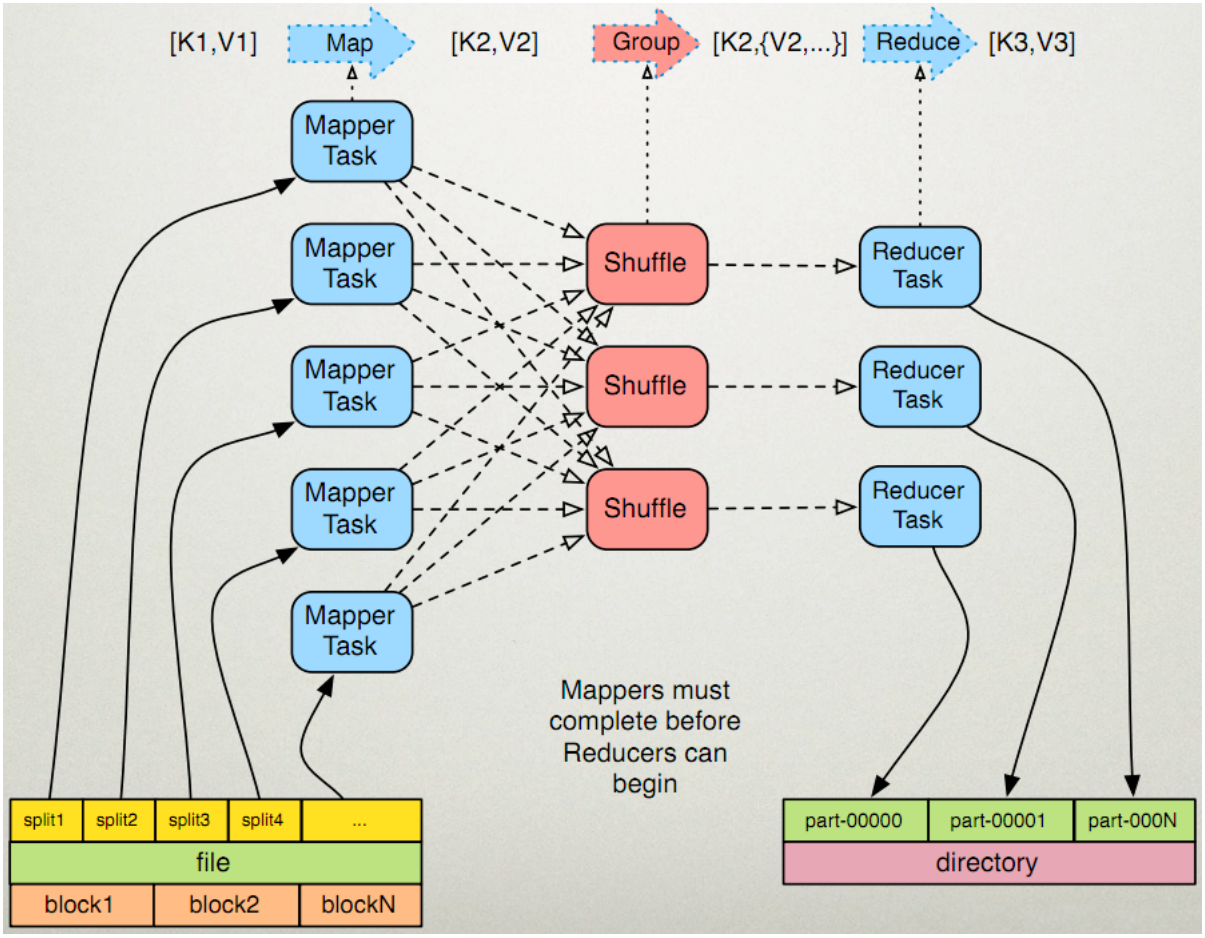
整个map的执行过程如下图所示：



map输出溢写（spill）&& Shuffle

- **Shuffle**

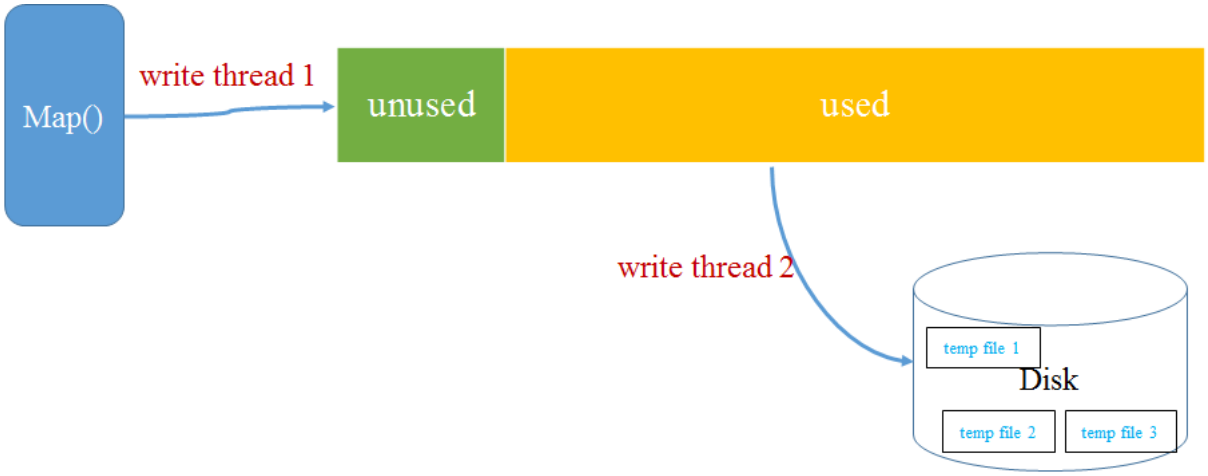
一般把从map任务输出到reducer任务输入之间的map/reduce框架所做的工作叫做shuffle。这部分也是map/reduce框架最重要的部分。下面将详细介绍这个shuffle中的各个步骤。



- **内存缓冲区**

Map/Reduce框架为InputSplit中的每个键值对调用一次 map(WritableComparable, Writable, OutputCollector, Reporter)操作，调用一次map()操作后就会得到一个新的（key,value)对。**当Map程序开始产生结果的时候，并不是直接写到文件的，而是写到一个内存缓冲区(环形内存缓冲区)。**每个map任务都有一个内存缓冲区，存储着map的输出结果，这个内存缓冲区是有大小限制的，默认是100MB（可以通过属性io.sort.mb配置）。

当map task的输出结果很多时，就可能会超过100MB内存的限制，所以需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为“spill”，中文可译为 溢写。**这个溢写是由单独线程来完成，不影响往缓冲区写map结果的线程。**



溢写线程启动时不应该阻止map的结果输出，所以整个缓冲区有个溢写的比例spill.percent（可以通过属性Io.sort.spill.percent配置），这个比例默认是0.8，也就是当缓冲区的数据已经达到阈值（buffer size * spill percent = 100MB * 0.8 = 80MB），**溢写线程启动，锁定这80MB的内存，执行溢写过程。**Map任务的输出结果还可以往剩下的20MB内存中写，互不影响，但如果缓冲区满了，Map任务则会被阻塞。那么为什么需要设置写入比例呢？达到一定比例后，由于写缓存和读缓存是可以同时并行执行的，这会降低把缓存数据腾空的时间，从而提高效率。

- **分区**

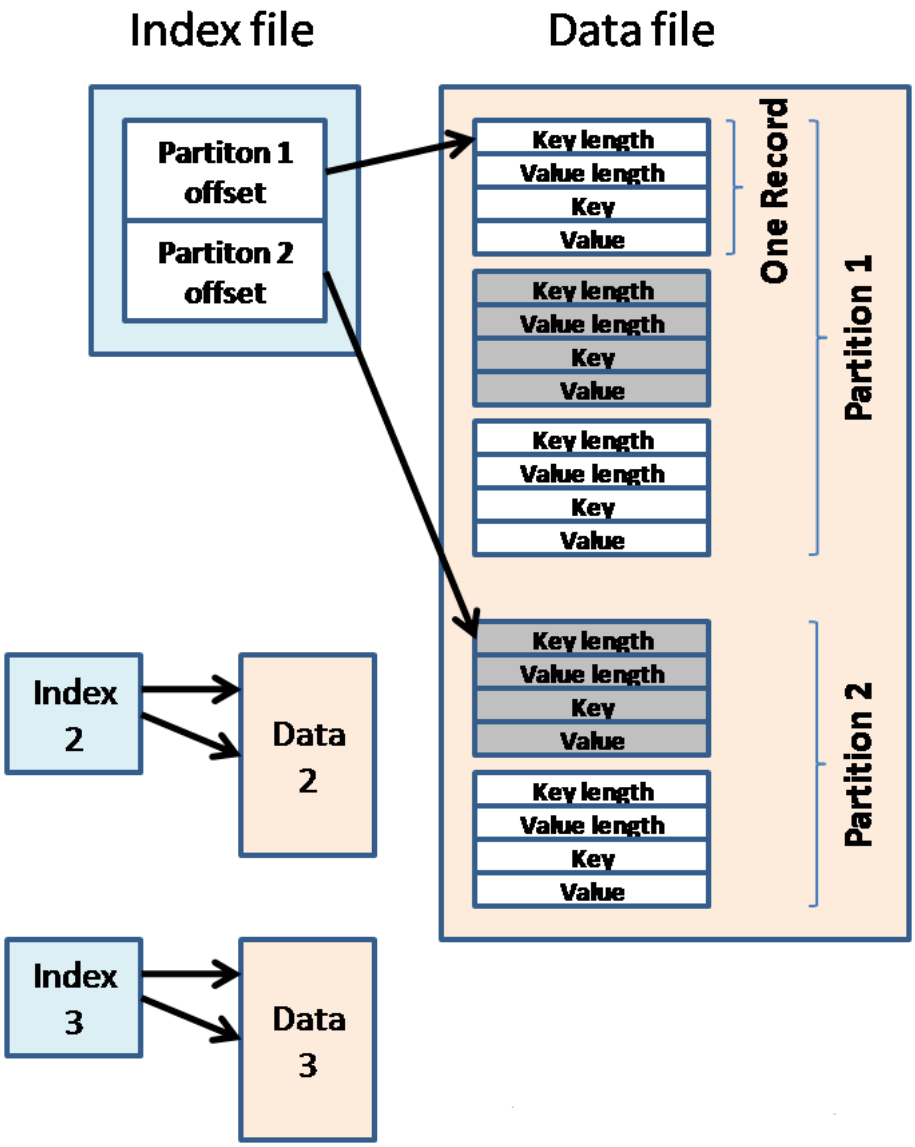
在把map()输出数据写入内存缓冲区之前会先进行Partitioner操作。Partitioner用于划分键值空间（key space）。MapReduce提供Partitioner接口，它的作用就是根据key或value及reduce的数量来决定当前的这对输出数据最终应该交由哪个reduce task处理。默认对key hash后再以reduce task数量取模。默认的取模方式只是为了平均reduce的处理能力，如果用户自己对Partitioner有需求，可以订制并设置到job上。

```
1  reducer=(key.hashCode() & Integer.MAX_VALUE) % numReduceTasks
```

HashPartitioner是默认的 Partitioner。

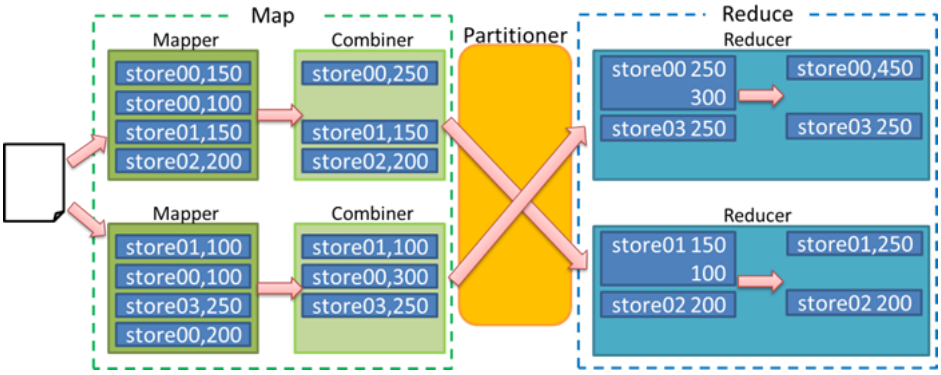
Partitioner操作得到的分区元数据也会被存储到内存缓冲区中。当数据达到溢出的条件时，读取缓存中的数据和分区元数据，然后把属与同一分区的数据合并到一起。对于每一个分区，都会在内存中根据map输出的key进行排序（排序是MapReduce模型默认的行为，这里的排序也是对序列化的字节做的排序），如果配置了Combiner，则排序后执行Combiner（Combine之后可以减少写入文件和传输的数据）。如果配置了压缩，则最终写入的文件会先进行压缩，这样可以减少写入和传输的数据。最后实现**溢出的文件内是分区的，且分区内是有序的。**

每次溢出的数据写入文件时，都按照分区的数值从小到大排序，内部存储是以tag的方式区分不同分区的数据；同时生成一个索引文件，这个索引文件记录分区的描述信息，包括：起始位置、长度、以及压缩长度，这些信息存储在IndexRecord结构里面。一个spill文件中的多个段的索引数据被组织成SpillRecord结构，SpillRecord又被加入进indexCacheList中。



- **Combiner**

Combiner最主要的好处在于减少了shuffle过程从map端到reduce端的传输数据量。



combiner阶段是程序员可以选择的，combiner其实也是一种reduce操作。Combiner是一个本地化的reduce操作，它是map运算的后续操作，主要是在map计算出中间文件前做一个简单的合并重复key值的操作，例如我们对文件里的单词频率做统计，map计算时候如果碰到一个hadoop的单词就会记录为1，但是这篇文章里hadoop可能会出现n多次，那么map输出文件冗余就会很多，因此在reduce计算前对相同的key做一个合并操作，那么文件会变小，这样就提高了宽带的传输效率，毕竟hadoop计算力宽带资源往往是计算的瓶颈也是最为宝贵的资源，但是combiner操作是有风险的，使

用它的原则是combiner的输入不会影响到reduce计算的最终输入，例如：如果计算只是求总数，最大值，最小值可以使用combiner，但是做平均值计算使用combiner的话，最终的reduce计算结果就会出错。

Combiner 也有一个性能损失点，因为它**需要一次额外的对于map输出的序列化/反序列化过程**。不能通过聚合将map端的输出减少到20-30%的话就不适用combiner。

- **压缩**

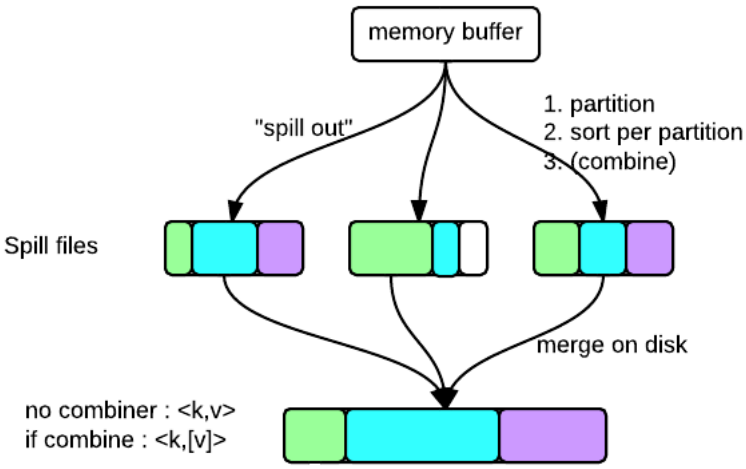
Map/Reduce框架为应用程序的写入文件操作提供压缩工具，这些工具可以为**map输出的中间数据和作业最终输出数据（例如reduce的输出）提供支持**。

压缩中间数据： 对map输出的中间数据进行合适的压缩可以减少map到reduce之间的网络数据传输量，从而提高性能。Lzo压缩格式是一个压缩map中间数据的合理选择，它有效利用了CPU。

压缩应用输出： 使用合适的压缩格式压缩输出数据能够减少应用的运行时间。Zlib/Gzip 格式在大多数情况下都是比较适当的选择，因为它在较高压缩率的情况下压缩速度也还算可以，bzip2 就慢得多了。

- **合并临时文件**

每次spill操作也就是写入磁盘操作时候就会写一个溢出文件，也就是说在做map输出有几次spill就会产生多少个溢出文件，等map输出全部做完后，map会合并这些输出文件生成最终的正式输出文件，然后等待reduce任务来拉数据。将这些溢写文件归并到一起的过程叫做Merge。

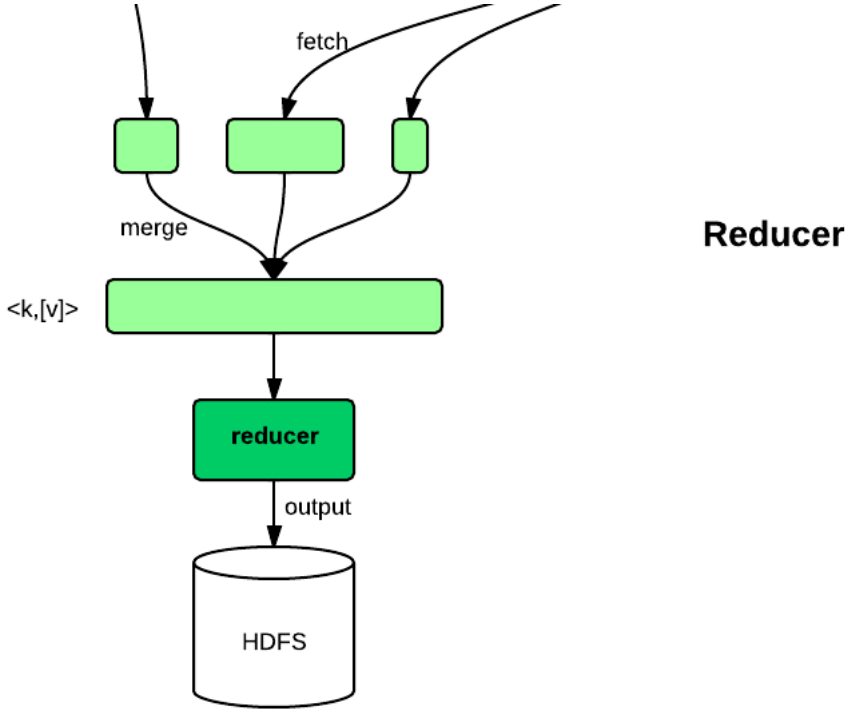


如果生成的文件太多，可能会执行多次合并，每次最多能合并的文件数默认为10，可以通过属性 `min.num.spills.for.combine` 配置。 多个溢出文件合并是，同一个分区内部也必须再做一次排序，排序算法是**多路归并排序**。是否还需要做combine操作，一是看是否设置了combine，二是看溢出的文件数是否大于等于3。最终生成的文件格式与单个溢出文件一致，也是按分区顺序存储，并且有一个对应的索引文件，记录每个分区数据的起始位置，长度以及压缩长度。这个索引文件名叫做file.out.index。

至此，map端的所有工作都已结束，最终生成的这个文件也存放在TaskTracker够得着的某个本地目录内。每个reduce task不断地通过RPC从JobTracker那里获取map task是否完成的信息，如果reduce task得到通知，Reduce就可以开始复制结果数据。

Reduce

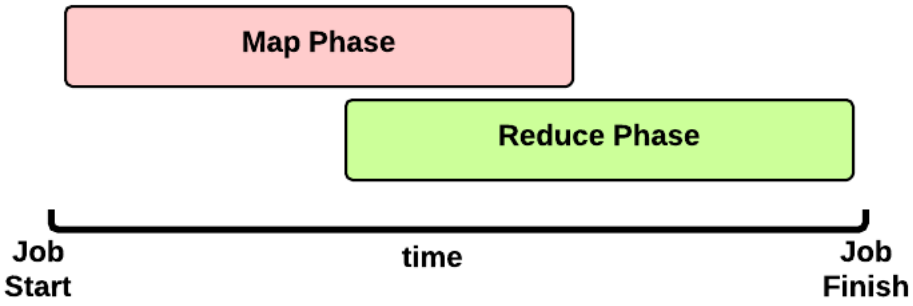
简单地说，reduce任务在执行之前的工作就是不断地拉取每个map任务的最终结果，然后对从不同地方拉取过来的数据不断地做merge，也最终形成一个文件作为reduce任务的输入文件。



reduce的运行可以分成copy、merge、reduce三个阶段，下面将具体说明这3个阶段的详细执行流程。

copy

由于job的每一个map都会根据reduce(n)数将数据分成map 输出结果分成n个partition，所以map的中间结果中是有可能包含每一个reduce需要处理的部分数据的。所以，为了优化reduce的执行时间，**hadoop中是等job的第一个map结束后，所有的reduce就开始尝试从完成的map中下载该reduce对应的partition部分数据**，因此map和reduce是交叉进行的，如下图所示：



reduce进程启动数据copy线程(Fetcher)，通过HTTP方式请求map task所在的TaskTracker获取map task的输出文件。由于map通常有许多个，所以对一个reduce来说，下载也可以是并行的从多个map下载，这个并行度是可以通过`mapred.reduce.parallel.copies（default 5）`调整。默认情况下，每个只会有5个并行的下载线程在从map下数据，如果一个时间段内job完成的map有100个或者更多，那么reduce也最多只能同时下载5个map的数据，所以这个参数比较适合map很多并且完成的比较快的job的情况下调大，有利于reduce更快的获取属于自己部分的数据。

reduce的每一个下载线程在下载某个map数据的时候，有可能因为那个map中间结果所在机器发生错误，或者中间结果的文件丢失，或者网络瞬断等等情况，这样reduce的下载就有可能失败，所以reduce的下载线程并不会无休止的等待下去，当一定时间后下载仍然失败，那么下载线程就会放弃这次下载，并在随后尝试从另外的地方下载（因为这段时间map可能重跑）。reduce下载线程的这个最大的下载时间段是可以通过`mapred.reduce.copy.backoff（default 300秒）`调整的。如果集群环境的网络本身是瓶颈，那么用户可以通过调大这个参数来避免reduce下载线程被误判为失败的情况。不过在网络环境比较好的情况下，没有必要调整。通常来说专业的集群网络不应该有太大问题，所以这个参数需要调整的情况不多。

merge

这里的merge如map端的merge动作类似，只是数组中存放的是不同map端copy来的数值。**Copy过来的数据会先放入内存缓冲区中**，然后当使用内存达到一定量的时候才刷入磁盘。这里需要强调的是，merge有三种形式：1)内存到内存 2)内存到磁盘 3)磁盘到磁盘。内存到内存的merge一般不适用，主要是内存到磁盘和磁盘到磁盘的merge。

这里的缓冲区大小要比map端的更为灵活，它基于JVM的heap size设置。这个内存大小的控制就不像map一样可以通过io.sort.mb来设定了，而是通过另外一个参数 `mapred.job.shuffle.input.buffer.percent（default 0.7）` 来设置， 这个参数其实是一个百分比，意思是说，shuffile在reduce内存中的数据最多使用内存量为：0.7 × maxHeap of reduce task。

也就是说，如果该reduce task的最大heap使用量（通常通过mapred.child.java.opts来设置，比如设置为-Xmx1024m）的一定比例用来缓存数据。默认情况下，reduce会使用其heapsize的70%来在内存中缓存数据。假设 `mapred.job.shuffle.input.buffer.percent` 为0.7，reduce task的max heapsize为1G，那么用来做下载数据缓存的内存就为大概700MB左右。这700M的内存，跟map端一样，也不是要等到全部写满才会往磁盘刷的，而是当这700M中被使用到了一定的限度（通常是一个百分比），就会开始往磁盘刷（刷磁盘前会先做sort）。这个限度阈值也是可以通过参数 `mapred.job.shuffle.merge.percent（default 0.66）`来设定。与map 端类似，这也是溢写的过程，这个过程中如果你设置有Combiner，也是会启用的，然后

在磁盘中生成了众多的溢写文件。这种merge方式一直在运行，直到没有map端的数据时才结束，然后启动磁盘到磁盘的merge方式生成最终的那个文件。

reducer

当reduce将所有的map上对应自己partition的数据下载完成后，就会开始真正的reduce计算阶段。当reduce task真正进入reduce函数的计算阶段的时候，有一个参数也是可以调整reduce的计算行为。也就是`mapred.job.reduce.input.buffer.percent（default 0.0）`。由于reduce计算时肯定也是需要消耗内存的，而在读取reduce需要的数据时，同样是需要内存作为buffer，这个参数是控制，需要多少的内存百分比来作为reduce读已经sort好的数据的buffer百分比。默认情况下为0，也就是说，默认情况下，reduce是全部从磁盘开始读处理数据。如果这个参数大于0，那么就会有一定量的数据被缓存在内存并输送给reduce，当reduce计算逻辑消耗内存很小时，可以分一部分内存用来缓存数据，反正reduce的内存闲着也是闲着。

Reduce在这个阶段，框架为已分组的输入数据中的每个 <key, (list of values)>对调用一次 `reduce(WritableComparable, Iterator, OutputCollector, Reporter)`方法。 Reduce任务的输出通常是通过调用 OutputCollector.collect(WritableComparable, Writable)写入 文件系统的。**Reducer的输出是没有排序的。**

那么一般需要多少个Reduce呢？

Reduce的数目建议是0.95或1.75乘以 (* mapred.tasktracker.reduce.tasks.maximum)。 用0.95，所有reduce可以在maps一完成时就立刻启动，开始传输map的输出结果。用1.75，速度快的节点可以在完成第一轮reduce任务后，可以开始第二轮，这样可以得到比较好的负载均衡的效果。

reduces的性能很大程度上受shuffle的性能所影响。应用配置的reduces数量是一个决定性的因素。太多或者太少的reduce都不利于发挥最佳性能: **太少的reduce会使得reduce运行的节点处于过度负载状态**，在极端情况下我们见过一个reduce要处理100g的数据。这对于失败恢复有着非常致命的负面影响，因为失败的reduce对作业的影响非常大。**太多的reduce对shuffle过程有不利影响。**在极端情况下会导致作业的输出都是些小文件，这对NameNode不利，并且会影响接下来要处理这些小文件的mapreduce应用的性能。在大多数情况下，应用应该保证每个reduce处理1-2g数据，最多5-10g。

The output files

作业的输出OutputFormat 描述Map/Reduce作业的输出样式。Map/Reduce框架根据作业的OutputFormat来：

```
1 1. 检验作业的输出，例如检查输出路径是否已经存在。
2 2. 提供一个RecordWriter的实现，用来输出作业结果。 输出文件保存在FileSystem上。
```

OutputFormat主要有以下几种：

OutputFormat:	Description
TextOutputFormat	Default; writes lines in "key \t value" form
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Disregards its inputs

OutputFormats provided by Hadoop

TextOutputFormat是默认的 OutputFormat。

计数器（Counters）

计数器（Counters）展现一些全局性的统计度量，这些度量由map/reduce框架本身，也可由应用来设定。应用可以自行定义任意的计数器并且在map或者reduce方法中更新它们的值。框架会对计数器的值做全局聚合。 计数器适合于追踪记录一些量不是很大，但是很重要的全局性信息。不应该用于一些粒度过细的信息统计。 使用计数器的代价非常昂贵，因为在应用的生命周期内JobTracker需要给每一个map/reduce任务维护一组计数器（定义了多少个就维护多少个）。

Reporter是用于map/reduce应用程序报告进度，设定应用级别的状态消息， 更新Counters（计数器）的机制。

Ref

- Apache Hadoop: Best Practices and Anti-Patterns
- Understanding Hadoop Clusters and the Network
- Introduction to MapReduce
- http://hadoop.apache.org/docs/r1.0.4/cn/streaming.html
- MapReduce:详解Shuffle过程
- http://hadoop.apache.org/docs/r1.0.4/cn/mapred_tutorial.html
- Anatomy of a MapReduce Job
- http://pennywong.gitbooks.io/hadoop-notebook/content/mapreduce/introduction.html
- https://developer.yahoo.com/hadoop/tutorial/module4.html
- https://developer.yahoo.com/hadoop/tutorial/module5.html

Posted by Jamzy Wang hadoop